AD-A041 625 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 12/1
A PROOF RULE FOR EUCLID PROCEDURES.(U)
MAY 77 J V GUTTAG, J J HORNING, R L LONDON DAHC15-72-C-0308
UNCLASSIFIED ISI/RR-77-60 NL

1 OF 1
ADA041625

END

DATE
FILMED
8 - 77

ARPA ORDER NO. 2223

ISI/RR-77-60
May 1977

John V. Guttag
University of Southern California

James J. Horning
University of Toronto

Ralph L. London
USC Information Sciences Institute

A Proof Rule for Euclid Procedures

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

ERRATUM

A PROOF RULE FOR EUCLID PROCEDURES

John V. Guttag, James J. Horning, and Ralph L. London

ISI/RR-77-60 May 1977

Page 2, line -7 should be

$$\text{assign}(x, \langle i_1, \ldots, i_{n-1} \rangle, \text{assign}(x(i_1) \ldots (i_{n-1}), i_n, y))$$

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) ISI/RR-77-60	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) A PROOF RULE FOR EUCLID PROCEDURES.	5. TYPE OF REPORT & PERIOD COVERED (7) Research Report. / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) John V. Guttag, Univ. of So. California; James J. Horning, Univ. of Toronto; Ralph L. London, ISI	8. CONTRACT OR GRANT NUMBER(s) (15) NSF-MCS76-06089; DAHC15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order No. 2223
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, VA 22209	(11) 12. REPORT DATE: 23 May 1977; 13. NUMBER OF PAGES: 12
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): (12) 14p.	15. SECURITY CLASS. (of this report): UNCLASSIFIED; 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

(OVER)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

proof rules, procedure-call rule, program verification, Euclid, axiomatic definition, aliasing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407 952

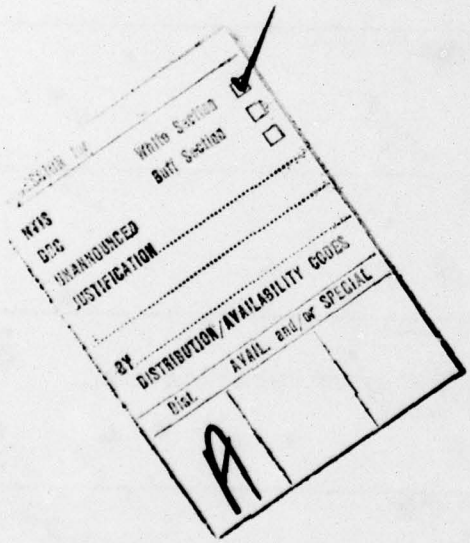
4B

18. SUPPLEMENTARY NOTES

This paper will be presented at the IFIP TC-2 Working Conference on Formal Description of Programming Concepts to be held in St. Andrews, New Brunswick, Canada, August 1-5, 1977. The proceedings will be edited by E. J. Neuhold and published by North-Holland Publishing Company.

20. ABSTRACT

The proof rules of Euclid, like the axiomatization of Pascal, presents a single definition of the various features of the language being defined. Little effort is made to explain the proof rules or to compare them to alternatives. In this paper we take one of our more complex proof rules, the rule for procedure definition and call, and attempt both to explain its workings and to compare it to two alternative rules. The rules we have chosen for comparison are Hoare's adaptation rule, from which our rule is derived, and the Pascal procedure-call rule.



A PROOF RULE FOR EUCLID PROCEDURES

John V. Guttag, University of Southern California

James J. Horning, University of Toronto

Ralph L. London, USC Information Sciences Institute

May 23, 1977

Abstract: The proof rules of Euclid, like the axiomatization of Pascal, presents a single definition of the various features of the language being defined. Little effort is made to explain the proof rules or to compare them to alternatives. In this paper we take one of our more complex proof rules, the rule for procedure definition and call, and attempt both to explain its workings and to compare it to two alternative rules. The rules we have chosen for comparison are Hoare's adaptation rule, from which our rule is derived, and the Pascal procedure-call rule.

Introduction and notation

An overriding goal in the design of Euclid [Lampson77, Popek77] was that Euclid programs be formally verifiable. While designing the language, it was therefore necessary to devote attention to the form its proof rules [London77] might take. In light of the difficulties encountered in writing proof rules for the procedure-call mechanisms of other languages, special attention was devoted to the design of this aspect of Euclid. In particular, the language was designed with the intent of using a rule similar to Hoare's rule of adaptation [Hoare71] as implemented in [Igarashi75].

The research described here was supported in part by the National Science Foundation Grant MCS76-06089, in part by a Research Leave Grant from the University of Toronto and a Grant from the National Research Council of Canada, and in part by the Defense Advanced Research Projects Agency Contract DAHC-15-72-C-0308. The views expressed are those of the authors. Horning is now affiliated with Xerox Palo Alto Research Center.

This paper will be presented at the IFIP TC-2 Working Conference on Formal Description of Programming Concepts to be held in St. Andrews, New Brunswick, Canada, August 1-5, 1977. The proceedings will be edited by E. J. Neuhold and published by North-Holland Publishing Company.

A key feature in the language design was the elimination of all aliasing. This allowed us to meet our design goal of relaxing the adaptation rule's proscription of global variables. The explicit appearance in the procedure heading of a list of all global variables imported into (i.e., used by) the procedure made it easier for us to deal with the "implicit parameters" discussed in the axiomatization of Pascal [Hoare73], a language to which the designers of Euclid owe a heavy debt.

Our notation for expressing proof rules is derived largely from that of [Hoare73]. P(y/x), which may be read "P with y for x," denotes the formula obtained by systematically substituting y for all free occurrences of x in the predicate P. If a conflict between free variables of y and bound variables of P is introduced, it is resolved by a systematic change of the latter variables. The notation

$$P(y_1/x_1, \ldots, y_n/x_n)$$

denotes simultaneous substitution for all occurrences of any x_i by the corresponding y_i; occurrences of x_i within any y_j are *not* replaced. The expressions $x_1, \ldots, x_n$ must be distinct; otherwise the simultaneous substitution is not defined.

Additional notation is needed to define substitution for an array reference. Borrowing from [McCarthy63], we use assign(x,i,y), where x is an array identifier, to stand for the array x with the element x(i) replaced by y. The formula

$$P(y/x(i)),$$

denoting a substitution for an array reference, is then defined to be

$$P(\text{assign}(x, i, y)/x).$$

To extend this definition to the substitution for a multiply-subscripted array reference, we define

$$\text{assign}(x, \langle i_1, \ldots, i_n \rangle, y), \text{ where } n \geq 2$$

to be equal to

$$\text{assign}(x, \langle i_1, \ldots, i_{n-1} \rangle, \text{assign}(x, i_n, y))$$

and the formula

$$P(y/x(i_1) \ldots (i_n)), \text{ where } n \geq 2$$

as equivalent to

$$P(\text{assign}(x, \langle i_1, \ldots, i_n \rangle, y)/x).$$

Substitution for a record field reference is defined analogously; for dereferenced pointer variables the identity $p\uparrow = C(p)$ is used where C is the collection associated with p.

The Euclid procedure rule

The components of a Euclid procedure declaration are given by the schema[1]

procedure p(var x, nonvar c) imports (var y, nonvar d) =
pre P; post Q; begin S end.

Basic symbols ("reserved words") of Euclid are underlined. Nonvar denotes the list of const and readonly identifiers.[2] The procedure named p is declared with explicit formal parameters x and c, imported identifiers y and d, and body S. Associated with each formal parameter declared in the procedure heading is its type. From the heading, we may, therefore, generate an assertion of the form

$$x \in \text{type of } x \wedge c \in \text{type of } c.$$

We will call this assertion H. Similar assertions for y and d are provided by the proof rules for declarations. P, the precondition for p, is a predicate involving x, c, y, and d; it gives the only relation among those four identifiers that may be assumed by S. Q is the postcondition, or specification of the result, of p. In addition to x, c, y, and d, Q may involve x' and y', fresh variables which denote the initial values of x and y at entry to S.

The basic proof rule defining a procedure call p(a, e) with actual parameters a and e corresponding to the formal parameters x and c is

$$\frac{H \vdash x{=}x' \wedge y{=}y' \wedge P \{ S \} Q}{P(a/x, e/c) \wedge (Q(a*/x, e/c, y*/y, a/x', y/y') \supset R(a*/a, y*/y)) \{ p(a, e) \} R}.$$

The premise of this proof rule simply requires that H |-- P { S } Q be shown--that is, *in terms of formal parameters*, assume P and show, using *all* of the Euclid proof rules and the assertion H, that the execution of S will yield Q. P { S } Q need be shown only once for each procedure declaration, not once for each call, because the rule "adapts" this single proof to every call (hence the name adaptation). The other two terms of the premise merely define the prime notation.

The conclusion of this proof rule states that it is sufficient for the predicate of the form $P \wedge (Q \supset R)$ to hold prior to the call of p(a, e), if the predicate R is to hold after the call (like the assignment axiom, though with quite different details). The term P in $P \wedge (Q \supset R)$ states that the actual parameters satisfy the precondition P of p, thereby discharging the assumption of P in the premise. The term $Q \supset R$ states that Q may be used in proving (the substituted) R. The simultaneous substitutions into P and Q are clearly well-defined because the formal parameters x, c, and y (and x' and y') are distinct. The substitution into R is well-defined because Euclid contains the restriction that all of the var parameters a and the imported var variables y of a call are distinct, i.e., non-overlapping. The latter restriction, which we will call the no-aliasing

---

[1] One may view x, c, y, and d as single variables or as lists (vectors) of variables; in either case the type information of the variables is omitted.

[2] No assignments may be made to these identifiers within the procedure p.

restriction, is a generalization of an adaptation rule restriction and exists in part to satisfy this proof rule.

To see how information after the call--namely Q and R--is carried back across the call of p, it is necessary to look at the substitutions in more detail. The symbol "*" indicates that the variables a and y (in Q and R) must be replaced by fresh variables. This is necessary because the names a and y in $P \wedge (Q \supset R)$ refer, as always, to values before the call of p. New variables must be invented for a and y that will refer (*before* the call) to the changed values of these variables after the call. No other variables need be invented, because only a and y can be changed by p. The purpose of the substitutions can now be seen as the replacement of formal by actual parameters and as the introduction of fresh variables for the actuals a and y. The substitutions of a for x' and y for y' reflect the fact that a and y are the initial values of the actual parameters and should replace x' and y'. While both the prime notation and the *s are used to refer at a point to values of variables not available at the point, they are different in two respects. First, primes involve formal parameters and *s involve actual parameters, and second, primes denote initial values and *s denote final values.

It should be noted that if two or more of the actuals, a, are components of the same array, a slight complication arises in substituting for the actuals. R(a*/a, y*/y) may evaluate to a formula of the form

$$R(a1*/B(i_1) \ldots (i_m), a2*/B(j_1) \ldots (j_n)).$$

Since the *no-aliasing restriction guarantees* that there exists a k where $1 \leq k \leq m \leq n$ such that $i_k \neq j_k$, the substitution is well-defined. Applying the rule for array element substitution will reduce this to

$$R(assign(B, <i_1, \ldots, i_m>, a1*)/B, assign(B, <j_1, \ldots, j_n>, a2*)/B).$$

At this point simultaneous substitution is no longer well-defined. We therefore define *extended simultaneous substitution* with the rule

$$R(assign(B, <i_1, \ldots, i_m>, a1*)/B, assign(B, <j_1, \ldots, j_n>, a2*)/B) =$$
$$R(assign(assign(B, <i_1, \ldots, i_m>, a1*), <j_1, \ldots, j_n>, a2*)/B).$$

Note that in verifying Euclid programs this situation can only arise in connection with substitutions generated by application of the procedure-call rule. In this environment, we know, by the no-aliasing restriction, that replacing the simultaneous substitutions with what is in effect a sequential substitution involves no information loss. Consider, for example, the substitution

$$P(x/A(i)(j), y/A(k)(m)(n)).$$

It first reduces to

$$P(assign(A, <i, j>, x)/A, assign(A, <k, m, n>, y)/A)$$

and then (by extended simultaneous substitution) to

P(assign(assign(A, <i, j>, x), <k, m, n>, y)/A).

Since we know that either $i \neq k$ or $j \neq m$, we know that the outer assignment will not obliterate any of the values imparted by the inner assignment. Without the no-aliasing restriction, this might not be the case. Whether with aliasing the resulting substitution accurately mirrors the effect of the procedure depends upon the order in which the formals are assigned to in the procedure. The problem is that an assignment to one formal may overwrite a previous assignment to a different formal. The sequential substitution defined by "assign" may coincidentally reflect this correctly, but there is no way to ensure this.

The above basic proof rule is still missing two features of Euclid procedures. The first is a return statement with the informal meaning that control should return immediately from the procedure currently being executed. The proof rule for return is merely the statement that the axiom

Q { return } false

may be used in showing P { S } Q for p. The rule is expressed by adding this axiom to the premise to the left of the |--. The Q in the axiom is the postcondition Q of p. The form of this axiom should be compared to the go to axiom

assertion at L { go to L } false

where L is the end of the procedure. Q is the "assertion at 'the end'" and return means go to "the end." Note that the axiom Q { return } false is not sound if there is a module variable instantiated in the procedure p. In this case some "finalization" code may be executed before exiting the procedure. This code might falsify Q.

The second missing feature is recursion. As with Hoare's presentation of the adaptation rule, the rule above will fail if p is called recursively, for there is then no way to show the premise term P { S } Q. The solution is identical to Hoare's: invoke the rule "recursively" (or "inductively on the depth of recursion") by adding to the premise to the left of the turnstile a copy of the rule's conclusion, which indicates that the rule itself may be used on all recursive calls in showing P { S } Q.

The proof rule that includes recursion and returns is, therefore,

$$\frac{\begin{array}{l}[\ P(a1/x, e1/c) \wedge (Q(a1*/x, e1/c, y*/y, a1/x', y/y') \supset R1(a1*/a1, y*/y))\ \{\ p(a1, e1)\ \}\ R1,\\ Q\ \{\ \underline{\text{return}}\ \}\ \text{false}, H\ ]\ \text{|--}\\ \qquad x{=}x' \wedge y{=}y' \wedge P\ \{\ S\ \}\ Q\end{array}}{P(a/x, e/c) \wedge (Q(a*/x, e/c, y*/y, a/x', y/y') \supset R(a*/a, y*/y))\ \{\ p(a, e)\ \}\ R}\ .$$

In the recursion clause R1 replaces R, and a1 and e1 replace a and e to allow different predicates and different actual parameters to be used in recursive calls.

Earlier we noted that Euclid requires of a procedure call that all of the actual var parameters and imported var variables be distinct. In the absence of var subscript expressions (and var dereferenced pointers), the restriction is easy to enforce because a simple equality check

of variable *names* suffices. As we have already pointed out, however, with var subscript expressions or var dereferenced pointers the equality check must involve *values*. Thus, to enforce the restrictions on procedure calls, it is necessary to show that certain values are distinct. In Euclid this responsibility is placed on the compiler; if it cannot succeed by itself, the compiler generates *legality assertions* sufficient to show that the variables are distinct. For the procedure call p(A(i)(j), A(k)(m)(n)), where both parameters are var, the legality assertion would be

$$i \neq k \vee j \neq m.$$

In addition to showing all other assertions, the verifier of a Euclid program must prove that each legality assertion holds at the point of call.

Comparison with other procedure rules

If we ignore the recursion clause, Hoare's adaptation rule may be expressed as

$$\frac{P\ \{\ S\ \}\ Q}{P(a/x,\ e/c) \wedge \forall a(Q(a/x,\ e/c) \supset R)\ \{\ p(a,\ e)\ \}\ R}\ .$$

Imported variables are forbidden in this rule. A minor difference in this rule and ours is our explicit use of the prime notation for initial values. Any other consistent notation will suffice, including keeping it implicit (as here) as part of the notation for the pre- and postconditions. An important difference, however, is the method of renaming variables, which Hoare's rule accomplishes by the for-all quantifier on the variable a (this is the sole purpose of the quantifier). One of the restrictions in using this adaptation rule is that no actual var parameter may appear in a const parameter, which is consistent with the fact that the scope of the quantifier is over all of the term $Q \supset R$; thus, if a var parameter violated the restriction, it would be renamed incorrectly by the quantifier. Our proof rule, on the other hand, removes this restriction by a more selective renaming of variables in the term $Q \supset R$.

In place of $P \wedge (Q \supset R)$ in either procedure-call rule, it is possible to have instead $(P \supset Q) \supset R$ as is done in [Morris71]. The latter precondition is weaker than the former (they differ when P is false and R is true, i.e., when R can be proved without the help of Q). In Euclid we have adopted the language design decision that the meaning of a procedure is undefined if its precondition does not hold at the point of call. Under this assumption $(P \supset Q) \supset R$ becomes $P \wedge ((P \supset Q) \supset R)$, which is equivalent to $P \wedge (Q \supset R)$. We believe this to be a cautious and convenient choice. However, by a suitable selection of pre- and postconditions, we can show that there is no loss of generality: It is always possible to choose the precondition to be the constant true, which always holds, and to choose the postcondition to be $P \supset Q$ where initial and final values must be carefully written in P. The nontrivial precondition has been moved into the postcondition.

Our procedure rule is also different from the treatment in [Hoare73], which uses two rules: one for procedure declarations and one for calls. In addition to the general postcondition Q, the declaration rule also uses functions f(x, c, y, d) and g(x, c, y, d) which map the initial values of the

formal parameters and imported identifiers x, c, y, and d onto the final values x and y after the completion of S. The functions f and g are defined by the statement that we may deduce the existence of f and g, satisfying the implication

$$P \supset Q(f(x, c, y, d)/x,\ g(x, c, y, d)/y)$$

for all values of the variables involved. Informally, this says that if P is true, then when S is finished, Q will hold for the values computed for x and y. Since Pascal has value parameters rather than Euclid's const parameters, the Pascal rules have the additional restriction that the value parameter c cannot occur free in Q. (There is no loss of power, because f and g can include c.)

The same f and g functions--with actual parameters substituted for formal parameters--are used in the procedure-call rule

$$R(f(a, e, y, d)/a,\ g(a, e, y, d)/y)\ \{\ p(a, e)\ \}\ R.$$

This rule resembles the assignment axiom; in fact, the call rule is equivalent to the *simultaneous* assignments

$$a := f(a, e, y, d)$$
$$y := g(a, e, y, d).$$

This procedure-call rule implies dynamic scoping of the variable y, i.e., y is from the calling environment. The informal reports for both Euclid and Pascal, however, specify Algol-like static scoping. Were it not for Euclid's scope restrictions, our rule would exhibit the same problem. Fortunately, the Euclid report states [Lampson77, p. 39], "If a routine [i.e., procedure or function] is imported into a scope S, every identifier which it imports must also be imported into S." When this is combined with the prohibition against declaring an identifier which is the same as an identifier imported into the scope, it becomes impossible to write a program whose behavior is dependent upon the choice of static or dynamic scoping.

Another slight problem with the above rule is the failure to "and" the term P(a/x, e/c) to the left side of the procedure-call rule (as we do) to discharge the premise of the implication defining f and g.

In the Pascal rule initial values may be considered part of the functions f and g, which also accomplish the still-necessary renaming of variables, since f(a, e, y, d) and g(a, e, y, d) serve as the new names. This form of renaming permits an actual var parameter to appear in a value parameter.

The Euclid rule combines into a single rule the relationship between a procedure declaration and calls of that procedure. While this may be pleasant theoretically, an actual implementation in a verifier of either the Euclid or Pascal rules would ignore this fact and require essentially the same verifications regardless. As already noted, the key idea is simply that each procedure is verified once per declaration and the conclusion(s) of the rule(s) or the axioms are used for each call. Finally, if the procedure is deterministic, the choice between the general postcondition Q and the (single-valued) functions f and g is largely a matter of taste because in general Q could be simply

$$x = f(x', c, y', d) \wedge y = g(x', c, y', d).$$

In Pascal the determinism issue does not arise because all procedures are deterministic. In Euclid, on the other hand, the ability to traverse levels of abstraction via modules makes it possible to write procedures that appear to be nondeterministic at the abstract level.

Examples using the Euclid rule

We shall illustrate our procedure-call rule using the example in Appendix 2 of [Hoare73]. Consider the trivial procedure, with only one *var* parameter, one *const* parameter, and no imported identifiers[3]

```
procedure p(var a:integer; const b:integer) = pre true; post a ≤ 2*b;
    begin var c:integer;
        c := 2*b;
        if a > c then a := c end if
    end.
```

Letting S stand for the body of this procedure, we can easily prove

$$\text{true } \{ S \} \ a \leq 2*b.$$

The invocation of this procedure will (in general) change the value of a in some manner dependent on the initial values of a and b (and, if present and referenced from within S, on values of imported identifiers).

Now the effect ot any call of p(z, w) is to change z such that $z \leq 2*w$; using the procedure-call rule we may validly conclude for any R that

$$\text{true} \wedge (z*\leq 2*w \supset R(z*/z)) \ \{ p(z, w) \} \ R.$$

In particular, R might be merely $z \leq 2*w$ or R might involve variables other than z and w if the call p(z, w) followed statements involving them. The properties of a call of p are contained in the postcondition $a \leq 2*b$, which can be derived only from the properties of the body of the procedure p.

The given postcondition is not the only property that can be proved about p. For example, the postcondition $a = \min(a', 2*b)$ could be used, where the prime denotes the initial value of a. Hence we may validly conclude that

$$\text{true} \wedge (z* = \min(z, 2*w) \supset R(z*/z)) \ \{ p(z, w) \} \ R.$$

As another example, we define

---

[3] Since it is not possible in Euclid to declare a variable to be of type integer, we should, strictly speaking, replace the type integer by, say, the standard type signedInt.

```
procedure p1(var n:integer; const k:integer) = imports (const j:integer)
    pre true; post n=n'+k+j+1;
    begin n := n+k+j+1 end.
```

Clearly we have

$$x=x' \wedge P \{ S \} Q,$$

i.e.,

$$n=n' \wedge \text{true} \{ n := n+k+j+1 \} n=n'+k+j+1.$$

To show

$$m=m_0 \{ p1(m,m) \} m=m_0+m_0+j+1,$$

we use the procedure-call rule to obtain

$$(*) \qquad \text{true} \wedge (m*=m+m+j+1 \supset m*=m_0+m_0+j+1) \{ p1(m, m) \} m=m_0+m_0+j+1.$$

Since $m=m_0$ implies the left side of (*), the consequence rule gives the desired result. Finally, to show

$$m=5 \wedge j=3 \{ p1(m,m) \} m=14,$$

we need only show, in view of the procedure-call rule and the rule of consequence,

$$m=5 \wedge j=3 \supset \text{true} \wedge (m*=m+m+j+1 \supset m*=14),$$

i.e.,

$$m=5 \wedge j=3 \wedge m*=2*m+3+1 \supset m*=14.$$

Conclusion

We have been concerned with the procedure mechanism, as a language feature, and its associated proof rules. We believe the full Euclid proof rule covers the complete Euclid procedure mechanism (with the exception of the problem, discussed above, of returns from procedures in which modules have been instantiated). This is hardly surprising, since Euclid procedures were designed to satisfy a predetermined form of proof rule--adaptation with certain constraints. Still, the mechanism is sufficiently powerful for tasks intended to be programmed in Euclid. Moreover, although constraints such as the import lists were included largely to aid verification, they were justified by other design considerations as well. We feel that even if verification is not considered, these restrictions make Euclid a better programming language.

Our experiences in using the Euclid proof rule to verify programs are limited. However, successful use of Hoare's adaptation rule convinces us that our rule is both practical and useful.

Acknowledgments

We would like to thank David Musser for numerous discussions on a wide range of issues, Paul Hilfinger for discussions on the recursion and return clauses, and Nancy Bryan for helpful editing of a previous draft.

References

[Hoare71] C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach," *Symposium on Semantics of Algorithmic Languages*, E. Engeler (ed.), Springer-Verlag, 1971 (pp. 102-116).

[Hoare73] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica*, 2, 4, 1973 (pp. 335-355).

[Igarashi75] Shigeru Igarashi, Ralph L. London, and David C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation," *Acta Informatica*, 4, 2, 1975 (pp. 145-182).

[Lampson77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the Programming Language Euclid," *SIGPLAN Notices*, 12, 2, February 1977.

[London77] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof Rules for the Programming Language Euclid," Technical Report, May 1977.

[McCarthy63] J. McCarthy, "Towards a Mathematical Science of Computation," *Proceedings of IFIP Congress 62*, C. M. Popplewell (ed.), North-Holland, 1963 (pp. 21-28).

[Morris71] James H. Morris, Jr., "Comments on 'Procedures and Parameters' [Hoare71]," Unpublished Note, *circa* 1971.

[Popek77] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London, "Notes on the Design of Euclid," *Proceedings of an ACM Conference on Language Design for Reliable Software*, *SIGPLAN Notices*, 12, 3, March 1977 (pp. 11-18).